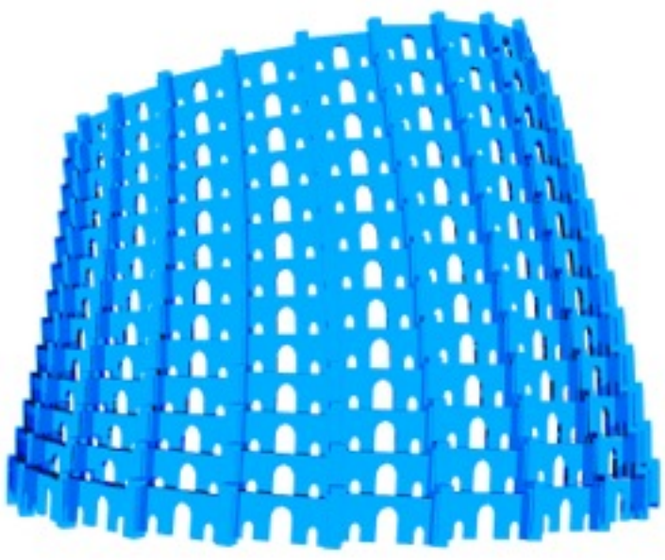
The promises and perils of mining git or GitHub

Hakan Aksu
University of Koblenz-Landau
Faculty of Computer Science
Software Languages Team



SOFTLANG

C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devenu

“The promises and perils of mining git”

MSR 2009

http://dblp.uni-trier.de/rec/html/conf/msr/BirdRBHGD09

RQ: What are the promises and perils of mining git?

Promise 1: Since any developer on a git project can make their own repository publicly accessible, it is possible to recover more history, including work in progress and work that never makes it into the stable codebase.

- Any person may make their own git repository publicly accessible

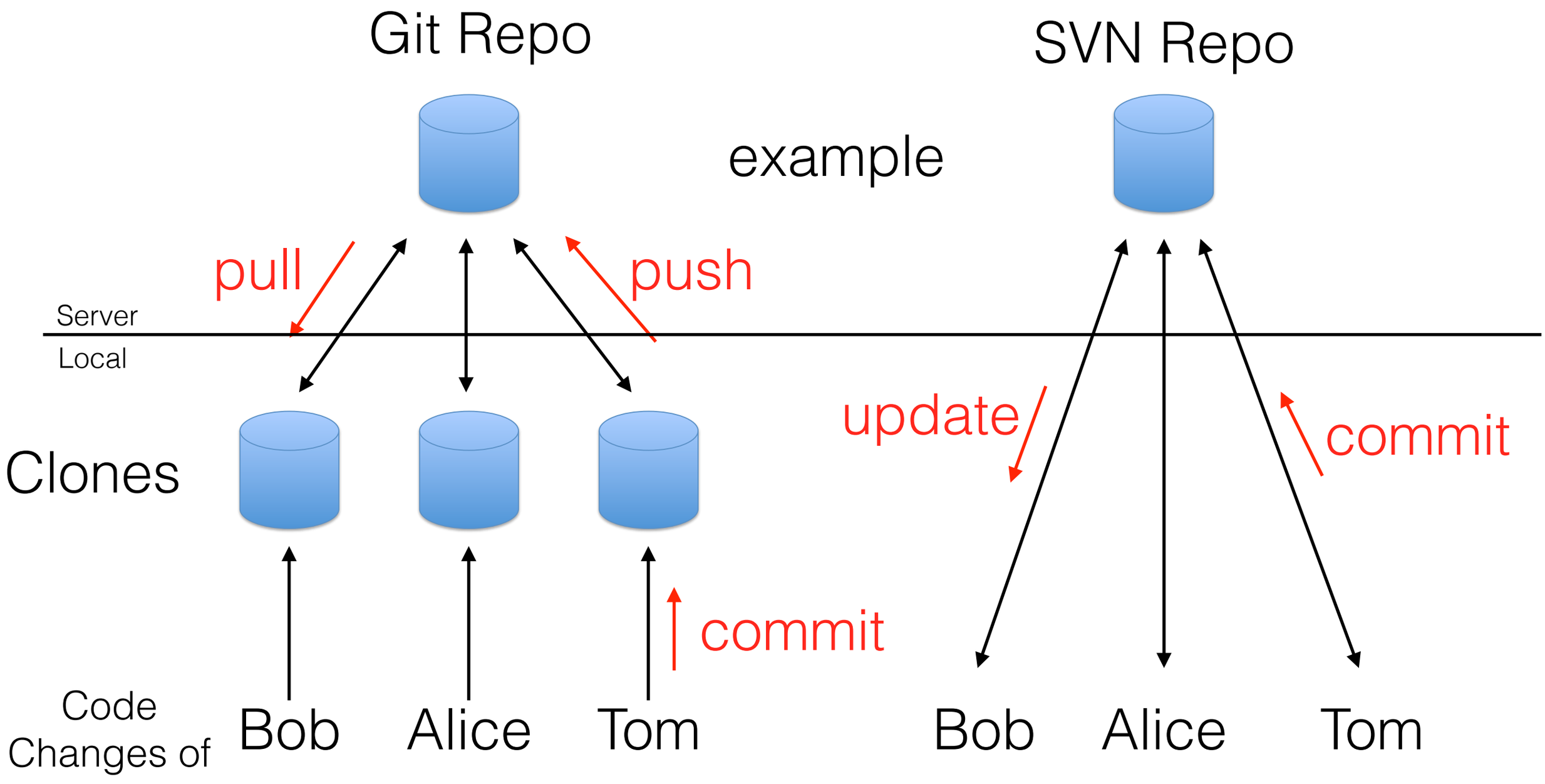
  e.g., ruby on rails has
  2009 -> 481 clones
  05/2016 -> 12641 clones

- Different developers can have different content in their repositories.

- It is possible to recover a more complete picture of the development process, including unpolished, experimental work that does not make it into the stable code base.

Peril 1: Git nomenclature differs from that of centralized SCMs (CSCMs): a) similar actions have different commands; and b) shared terms can have different meanings.

SCM = source code management

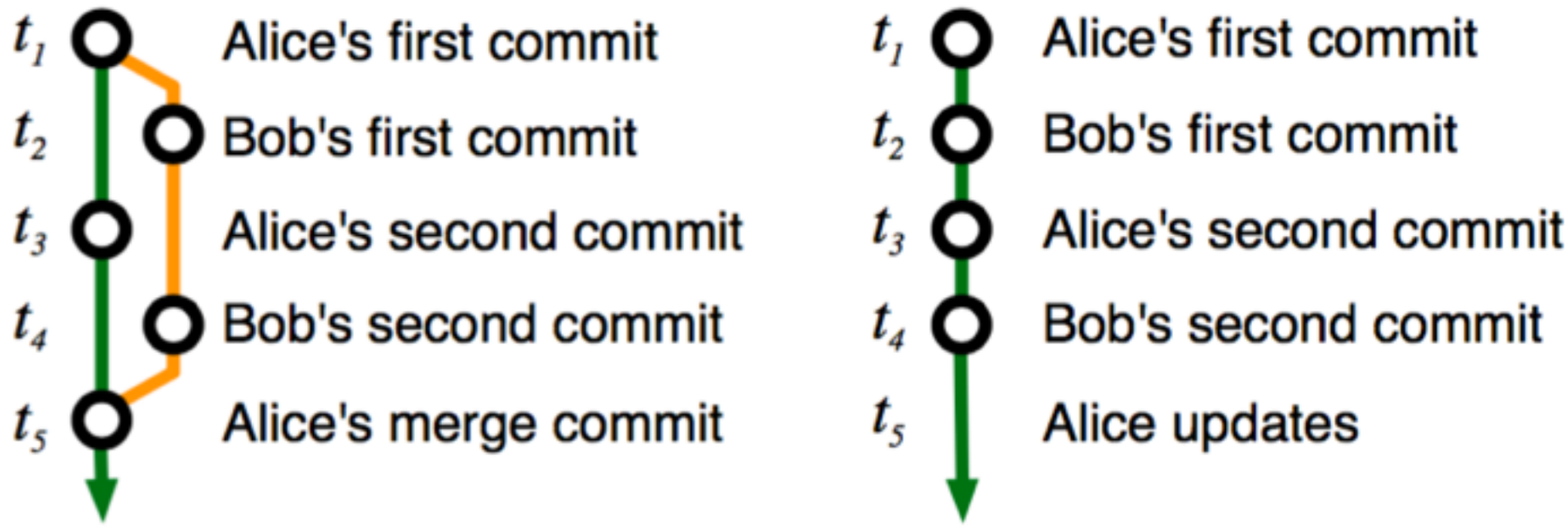


Peril 1: Git nomenclature differs from that of centralized SCMs (CSCMs): a) similar actions have different commands; and b) shared terms can have different meanings.

SCM = source code management

- git records information about both branches and merges
- the history of a repository is represented by a graph of commits
- In git a commit may have more than one parent (due to merging) and may be the parent of more than one commit (due to branching)
- Since no commit may be its own ancestor, the graph represented by commits and their directed parent-child relationships is a directed acyclic graph (DAG)
- Two git commits in different repositories have the same SHA-1 iff they have the same ancestors in the DAG and the same contents, which means that their working copies were identical when they were created.

Peril 2: Here Be “Implicit Branches!”



(a) Implicit Branching (b) SVN Linear History

Figure 3. The result of identical commit sequences.

Promise 2: Git facilitates recovery of richer project history through commit, branch and merge information in the DAGs of multiple repositories.

- Since git tracks both implicit and explicit branches and merges within and between repositories, it holds the promise of data not tracked by SVN (which tracks branches within the central repository, but not merges). This includes:

1. Implicit branches, showing how often developers pull and push changes from other repositories;

2. Feature (explicit) branches, showing collaboration activity: changes pulled directly between developers, vs. via an intermediary “official” repository.

3. Merge points, including the set of conflicted files, and who/when performed the merge and resolution.

4. Pulls from remote repositories, and the overall topology of the “pull network”.

5. The DAG, and set of commits in different repositories for the same project can determine the differences and “distance” from each other.

Peril 3: Git has no mainline, so analysis methods must be suitably modified to take the DAG into account.

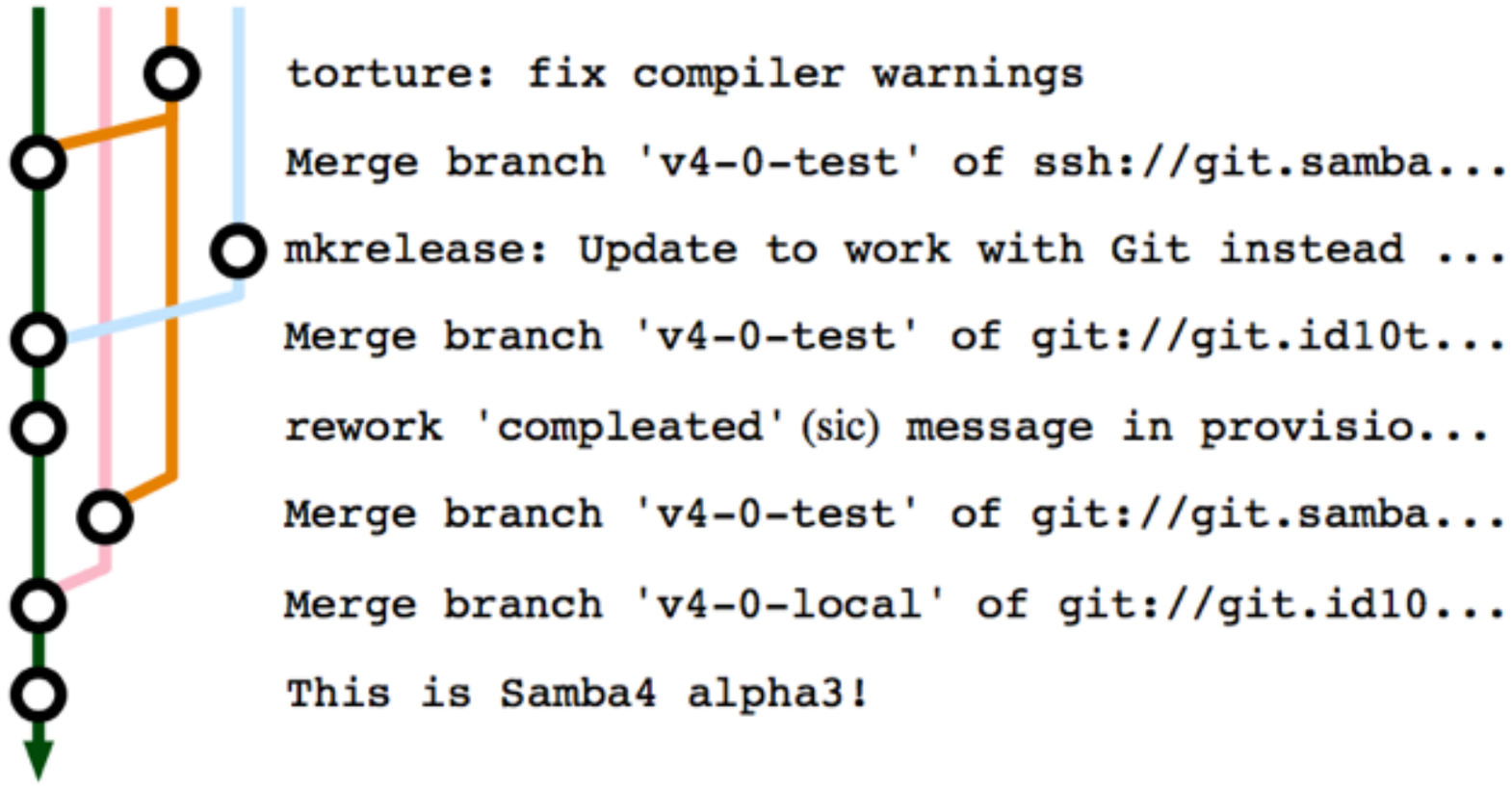


Figure 4. A subgraph of the git DAG of commits leading up to release of 4.0 alpha 3 of Samba.

Peril 4: Git history is revisionist: a repository owner can rewrite it.

- Git allows a user to rewrite history —> rebasing

- A user selects a sequence of commits to rebase he may then…

  - alter the order of commits

  - remove commits

  - squash the edits in multiple commits into one commit

  - flatten a sequence of commits on multiple branches onto a single branch.

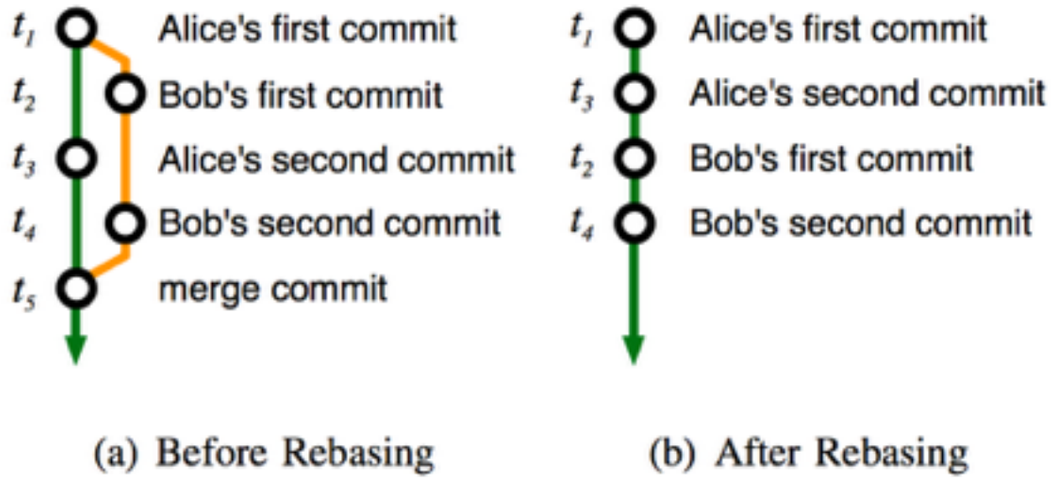


Figure 5. Example of Rebasing

Peril 5: You cannot always determine what branch a commit was made on.

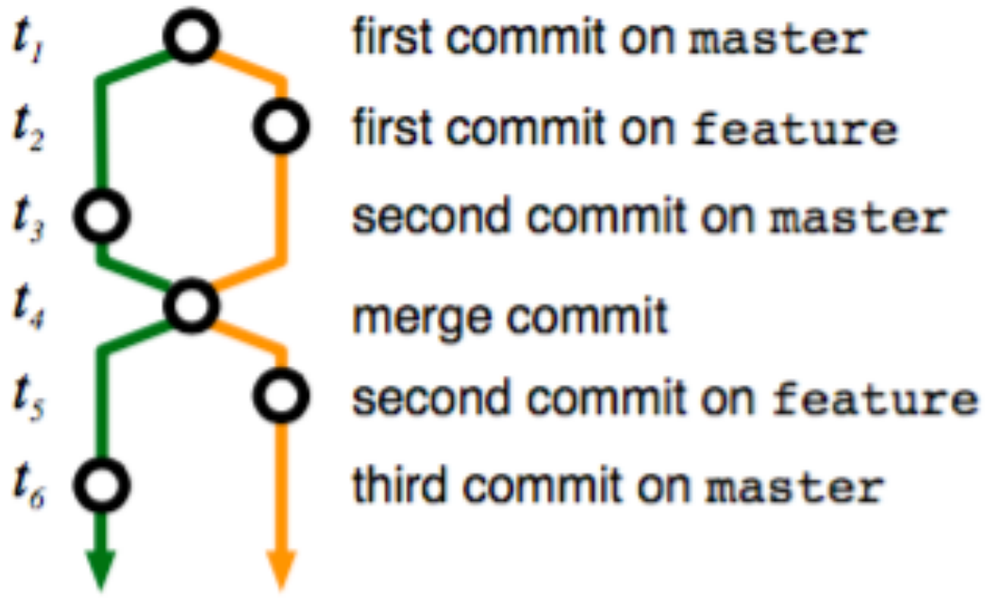


Figure 6. The git DAG after branches `master` and `feature` are each merged with each other.

- Alice merges a branch named *feature* to *master*, then *master* into *feature*. These merges share a single merge commit.

- If the repository were mined at time t6, it usually is not possible to tell which commits prior to the merge were made on *master* (green-line before t4) and which were made on *feature* (orange-line before t4)

Peril 6: It is not always possible to track the source of a merge or even determine if a merge occurred.

- Typically, when a developer pulls commits from some branch in a remote git repository to his local repository, the branch must be merged into the current local working branch. We'd like to know the source of that merge, both in terms of the git repository and branch within that repository.

- By default, git creates a log message for the merge commit with one of the following forms. Text in brackets may not always appear.

  - *Merge branch 'branchname' [into branch_name]*
    Merge [branch 'branchname' of] remote_repo_url

Peril 6: It is not always possible to track the source of a merge or even determine if a merge occurred.

- *The situations where the merge source is not available in the commit messages:*

    1. *If a merge of two branches results in a fast forward merge, no merge commit is created and thus no log message that contains the string will exist in the log.*

    2. *If there are conflicts during a merge, the developer will resolve the conflicts and commit their resolution with a log message that may not include the default text.*

    3. *If a developer rebases a series of commits which contain branches and merges, the result may be "flattened" and not include the merge commit.*

    4. *If a developer amends the commit message of a merge commit to something other than the default merge message (this is unlikely, but possible).*

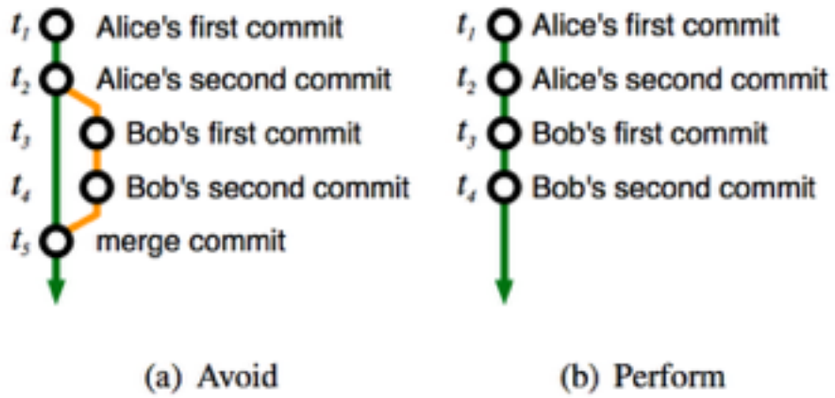


Figure 7. Alice's repository if git was told explicitly to avoid a fast forward merge (a) and normally, when performing a fast forward merge (b) after pulling Bob's changes.

Promise 3: Git records the information needed to correct Perils 3–6 in private logs.

- Git stores information about fast-forward merges, rebases and pulls in a logs directory (only local)

- The miner must have access to the private repository of each developer whose logs she wishes to mine.

Peril 7: The accessible data may only contain commits that are success selected.

- In one workflow, a developer creates a branch that contains changes to be pulled and reviewed by other developers.

- a commit's review history is not directly reflected in git

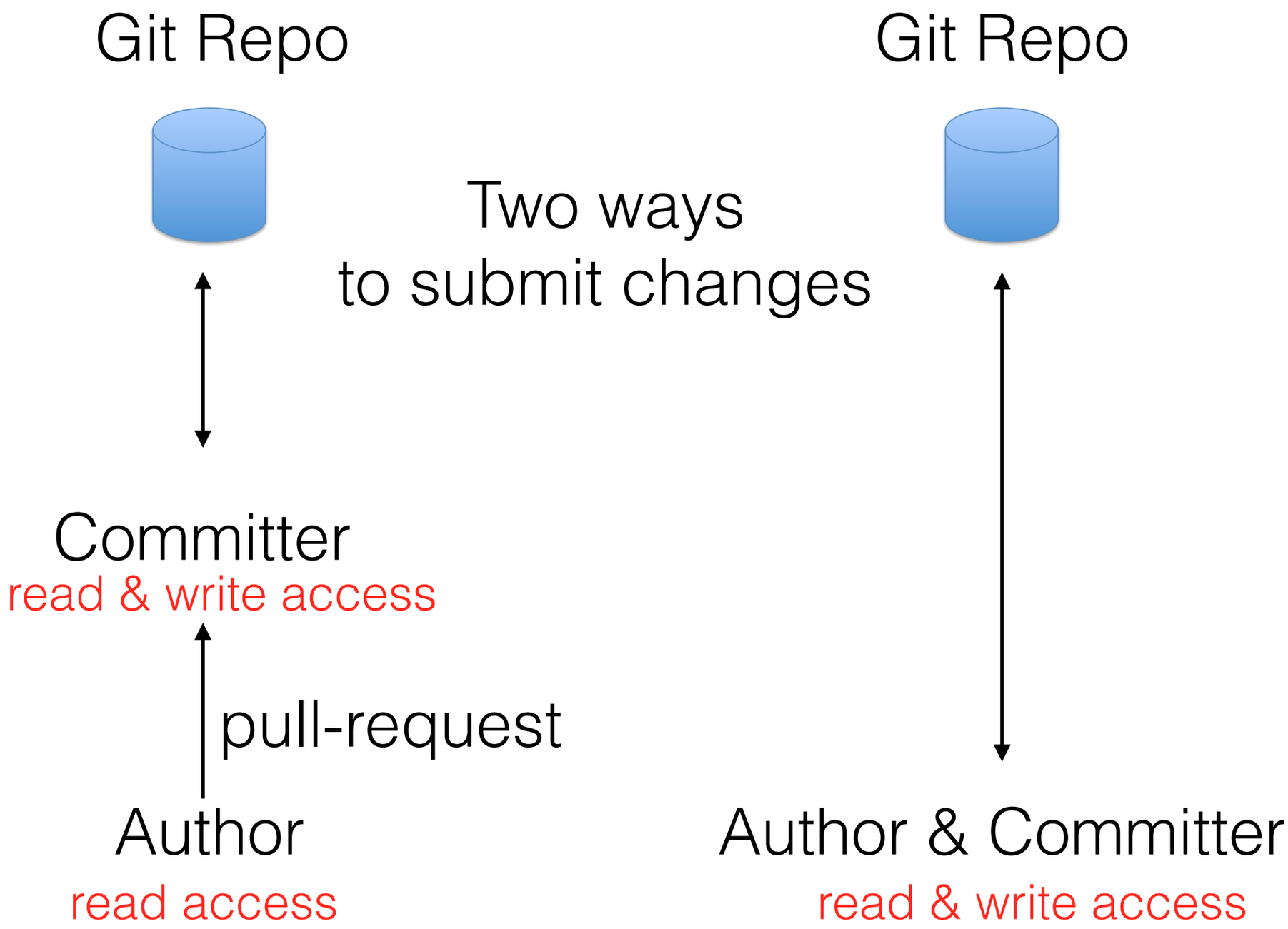
Promise 4: The signed-off-by and other attributes create a “paper trail.”

- In response to IP infringement allegations made by SCO, Linus Torvalds added a facility for people to “*sign off*” on a commit by adding a line such as *signedoffby John Doe <jdoe@foobar.com>* to the end of a commit message.

- There are also other attributes:
  - ackedby
  - Cc
  - reportedby
  - reviewedby
  - testedby

- These attributes are explicitly added to commits using the -s flag and append a line to the commit log message using a standard format that is easy to parse. A commit may have multiple fields appended as it moves from repository to repository and is reviewed and tested.

Promise 5: Git explicitly records authorship information for contributors who are not part of the core set of developers.



Promise 6: In git all metadata, notably history, is local.

- When a developer creates a local repository based on a pre-existing repository all information from the remote repository is transferred to the local repository

Promise 7: Git tracks content, so it can track the history of lines as they are moved or copied.

- Git is able to tell automatically if a file is renamed because the content remains the same and the history follows the content

- If svn blame is run to show who introduced each line, Bob would be indicated as the author of each line in the function sub. However, git blame -C -M, where -C finds code copies and -M finds code movement, would indicate that Alice had written each line of the file.

- Git also tracks text that moves between files and text that is copied multiple times as long as the number of lines moved is above a certain threshold (we observe the threshold to be around 4 or 5).

```
#include <math.h>
int add(int a)
{
...
}

int sub(int a)
{
...
}

void main(){...}
```

(a) original by Alice

```
#include <math.h>
int sub(int a)
{
...
}

int add(int a)
{
...
}

void main() {...}
```

(b) modified by Bob

Figure 9. Two versions of a file

Promise 8: Git is faster and often uses less space than centralized repositories

- The metadata is on the local machine, accessing a log or a diff executes locally, with no network latency.

- Git stores full (compressed) versions of files, rather than a sequence of diff s. This means that checking out a file is a constant- time operation, regardless of history.

- Checking out commits in an arbitrary (non– consecutive) order with git took 1–2 minutes per commit compared to 7–25 minutes with CVS

- The entire Mozilla repository weighs in at 12 GB when stored in SVN. However, when the repository was imported into git, that shrunk to 420 MB

Promise 9: Most SCMs such as CVS, SVN, Perforce and Mercurial (Hg), can be converted to git with the history of branches, merges and tags intact.

- By being able to convert nearly all repositories to git format, we have been able to reap some of the benefits of git, such as better origin detection, performance, and local copies of all information.

- We only need to write mining and analysis tools for one format rather than many.

E.Kalliamvakou, G.Gousios, K.Blincoe, L.Singer,
D.M.Germán and D. Damian,

“The promises and perils of mining github”

MSR 2014

http://dblp.uni-trier.de/rec/html/conf/msr/KalliamvakouGBSGD14

RQ: What are the promises and perils of mining GitHub for software engineering research?

Peril I: A repository is not necessarily a project.

- The project's main repository is not writable by potential contributors.
- Instead, these contributors fork (clone) the repository and make their changes independent of each other.
- They create a pull request, which specifies a local branch to be merged with a branch in the main repository.
- two types of repositories:
  - base repositories (those that are not forks)
  - forked repositories
- A project is a base repository with its forked repositories
- When a commit is made and is pulled into another repository via a GitHub pull-request, this commit does not appear in the history of the recipient repository; it only appears in the repository where the commit orginated. Therefore, measuring the activity of a repository independently of its forked repositories will ignore the activity of all of them as part of a single project.
- **Peril Avoidance Strategy: To analyze a project hosted on GitHub, one must consider the activity in both the base repository and all associated forked repositories.**

Peril II: Most projects have very few commits.

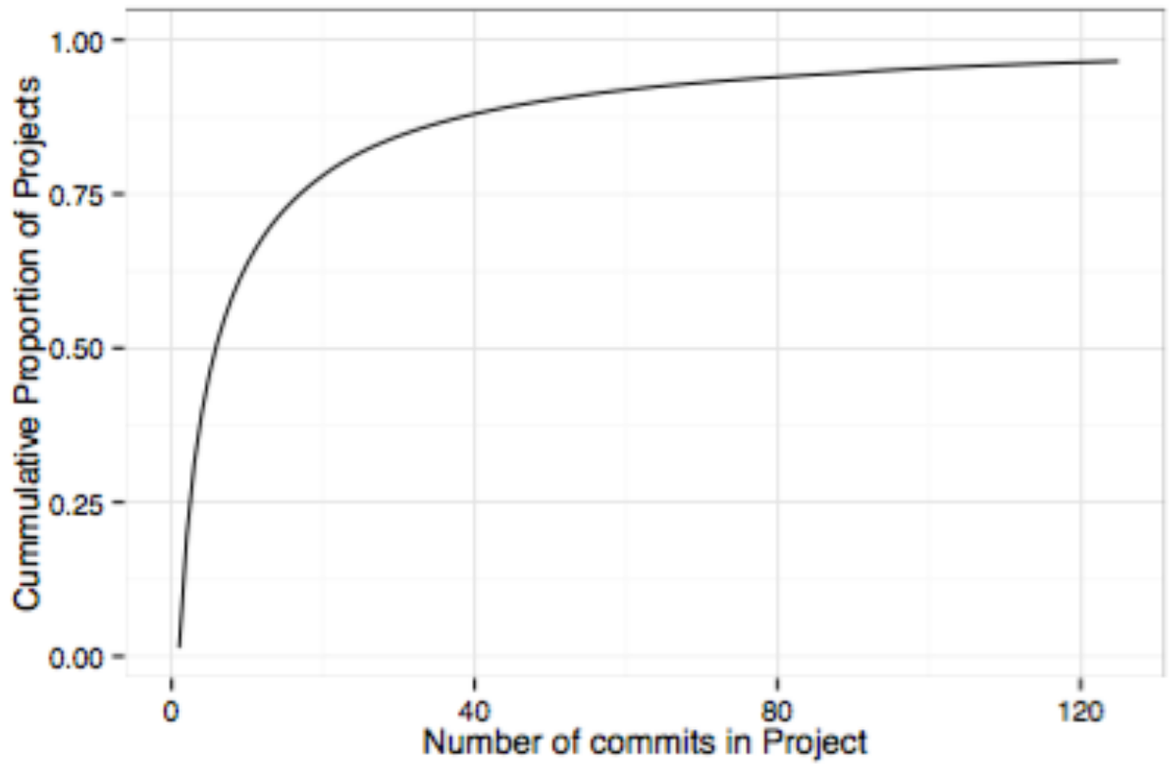


Figure 1: Cumulative ratio of projects with a given number of commits. Most projects have very few commits. The median number of commits per project is 6, and 90% of projects have less than 50 commits.

Peril III: Most projects are inactive.

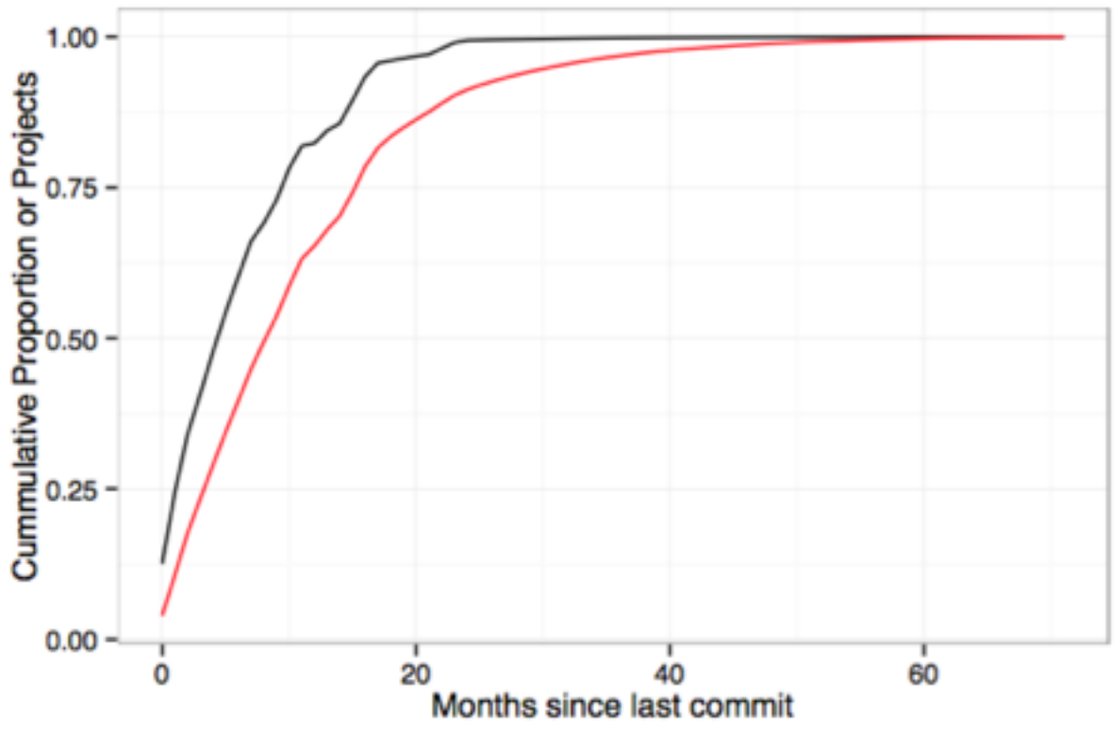


Figure 3: Cumulative ratio of active projects during the last n months since Jan 9, 2014. The red line is the proportion of projects created during the last n months. Approximately 46% of projects have been inactive in the last six months. Only 13% of projects were active in the last month, and 1/3 of them were created during that period.

- **Peril Avoidance Strategy: To identify active projects, consider the number of recent commits and pull requests.**

Peril IV: A large portion of repositories are not for software development.

Table 2: Number of repositories per type of use for the manual inspection. These categories are mutually exclusive.

Category of use	Number of repositories
Software development	275 (63.4%)
Experimental	53 (12.2%)
Storage	36 (8.3%)
Academic	31 (7.1%)
Web	25 (5.8%)
No longer accessible	11 (2.5%)
Empty	3 (0.7%)

- 34 of our 240 respondents (14%) said they use GitHub repositories for experimentation, hosting their websites, and for academic/class projects

- About 10% of respondants use GitHub specifically for storage.

- **Peril Avoidance Strategy: When selecting projects to analyze, one should not rely only on the types of files within their repositories to identify software development projects. Researchers should review the description and README file to ensure the project fits their research needs.**

Peril V: Two thirds of projects (71.6% of repositories) are personal.

- 72% of repositories have one committer, 91% have 2 or less, and 95% 3 or less.

- **Peril Avoidance Strategy: To avoid personal projects, the number of committers should be considered.**

Promise I: GitHub provides a valuable source of data for the study of code reviews in the form of pull requests and the commits they reference.

- GitHub made the “Fork & Pull” development model popular, but pull requests are not unique to GitHub; In fact, git includes the git-request-pull utility, which provides the same functionality at the command line. GitHub and other code hosting sites improved this process significantly by integrating code reviews, discussions and issues, thus effectively lowering the entry barrier for casual contributions. Combined, forking and pull requests create a new development model, where changes are pushed to the project maintainers and go through code review by the community before being integrated.

Peril VI: Only a fraction of projects use pull requests. And of those that use them, their use is very skewed.

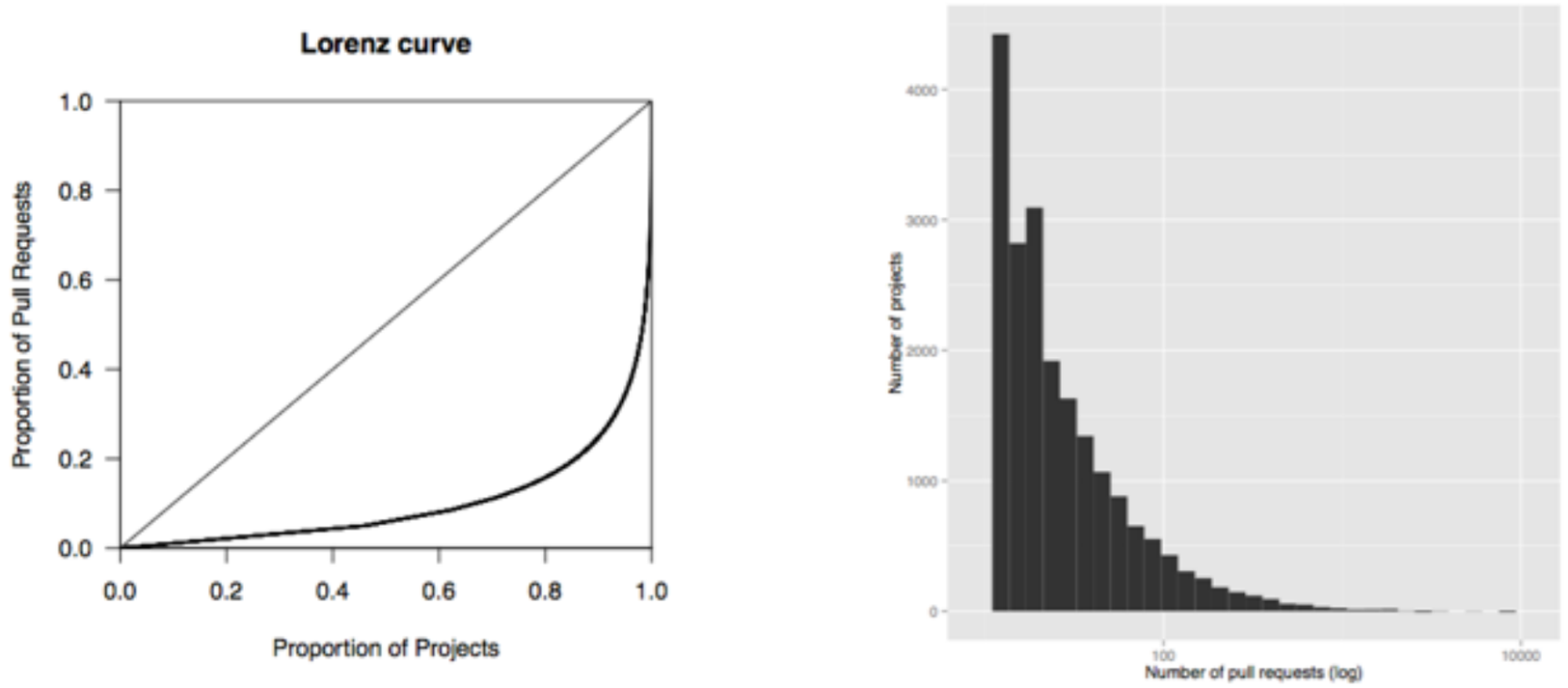


Figure 5: Lorenz curve for the number of pull requests per project (left) and the corresponding histogram (right). The top 1.6% of projects use 50% of the total pull requests. These plots only include projects with at least one pull request.

- The use of pull requests is not very widespread

- **Peril Avoidance Strategy: When researching the code review process on GitHub, the number of pull requests must be considered when selecting appropriate projects.**

Peril VII: If the commits in a pull-request are reworked (in response to comments) GitHub records only the commits that are the result of the peer-review, not the original commits.

- The set of commits that were reviewed might not be readily observable and might require further processing to recover them

- It is common in projects to require a commit squash (merging all different commits into a single one) before the set of commits is merged with the main repository.

- While GitHub does record the intermediate commits, it does not report them through its API as part of the pull request. Moreover, the original commits are deleted if the source repository is deleted. This means that at the time of analysis, the researcher can only observe the latest commit, which is the outcome of the review process.

- **Peril Avoidance Strategy: To perform analysis on the full set of commits involved in a code review, researchers must not rely on the commits reported by GitHub.**

Peril VIII: Most pull requests appear as non-merged even if they are actually merged.

- The versatility of git and GitHub enables at least three merging strategies:
  - Through GitHub facilities, using the merge button using git, by merging the main repository branch and the pull request branch. A variation of this merge strategy is cherry-picking, where only hand selected commits from the pull request branch are merged
    to the main branch.
  - By creating a textual patch between the pull request
  - and main repository branches and applying to the master branch. This is also known as commit squashing.

- If a project's policy is to only merge using *git*, all pull requests will be recorded as unmerged in GitHub. In practice however, most projects use a combination of GitHub and git merge strategies.

- **Peril Avoidance Strategy: Do not rely on GitHub's merge status, rather consider using heuristics like the ones described above to improve merge detection when analyzing merged pull requests.**

Promise II: The interlinking of developers, pull requests, issues and commits provides a comprehensive view of software development activities.

- For each opened pull request, an issue is opened automatically

- both issues and pull requests can be linked to repository-specific milestones, which helps projects to track progress.

- opportunity for very detailed studies of developer activity

Peril IX: Many active projects do not conduct all their software development in GitHub.

Table 3: Repositories hosted on GitHub labelled as mirrors. GitHub hosts mirrors from many sources, including SourceForge and Bitbucket. The bottom section shows subsets of the top section. Regular expressions are case-insensitive.

Set	Used regular expresion	No. Projets	No. Repos
Mirror Of	*mirror of .*repo|git mirror of*	1,851	12,709
Subsets			
Located at Sourceforge	*sourceforge|sf\.net*	117	511
Located at Bitbucket	*bitbucket*	91	249
From subversion repos	*\W(svn|subversion)\W*	622	4966
From mercurial repos	*\W(mercurial|hg)\W*	113	590
From CVS repos	*\Wcvs\W*	55	212

- 23% of committers or authors of a commit are not GitHub users
- Mirrors are replicas of the code hosted in another repository. In some cases, a mirror project clearly indicates that GitHub is not to be used for submission of code.
- **Peril Avoidance Strategy: Avoid projects that have a high number of committers who are not registered GitHub users and projects which explicitly state that they are mirrors in their description.**

Summary

Peril 1: Git nomenclature differs from that of centralized SCMs (CSCMs): a) similar actions have different commands; and b) shared terms can have different meanings.
Peril 2: Here Be "Implicit Branches!"
Peril 3: Git has no mainline, so analysis methods must be suitably modified to take the DAG into account.
Peril 4: Git history is revisionist: a repository owner can rewrite it.
Peril 6: It is not always possible to track the source of a merge or even determine if a merge occurred.
Peril 7: The accessible data may only contain commits that are success selected.

Promise 1: Since any developer on a git project can make their own repository publicly accessible, it is possible to recover more history, including work in progress and work that never makes it into the stable codebase.
Promise 2: Git facilitates recovery of richer project history through commit, branch and merge information in the DAGs of multiple repositories.
Promise 3: Git records the information needed to correct Perils 3–6 in private logs.
Promise 4: The signed-off-by and other attributes create a "paper trail."
Promise 5: Git explicitly records authorship information for contributors who are not part of the core set of developers.
Promise 6: In git all metadata, notably history, is local.
Promise 7: Git tracks content, so it can track the history of lines as they are moved or copied.
Promise 8: Git is faster and often uses less space than centralized repositories
Promise 9: Most SCMs such as CVS, SVN, Perforce and Mercurial (Hg), can be converted to git with the history of branches, merges and tags intact.

Peril I A repository is not necessarily a project.
Peril II Most projects have very few commits.
Peril III Most projects are inactive.
Peril IV A large portion of repositories are not for software development.
Peril V Two thirds of projects (71.6% of repositories) are personal.
Peril VI Only a fraction of projects use pull requests. And of those that use them, their use is very skewed.
Peril VII If the commits in a pull-request are reworked (in response to comments) GitHub records only the commits that are the result of the peer-review, not the original commits.
Peril VIII Most pull requests appear as non-merged even if they are actually merged.
Peril IX Many active projects do not conduct all their software development in GitHub.

Promise I: GitHub provides a valuable source of data for the study of code reviews in the form of pull requests and the commits they reference.
Promise II: The interlinking of developers, pull requests, issues and commits provides a comprehensive view of software development activities.